# JSExplain: A Double Debugger for JavaScript

Arthur Charguéraud
Inria & Université de Strasbourg, CNRS, ICube UMR 7357
arthur.chargueraud@inria.fr

Alan Schmitt
Inria & Univ Rennes, CNRS, IRISA
alan.schmitt@inria.fr

Thomas Wood
Imperial College London
thomas.wood09@imperial.ac.uk

## ABSTRACT

We present JSExplain, a reference interpreter for JavaScript that closely follows the specification and that produces execution traces. These traces may be interactively investigated in a browser, with an interface that displays not only the code and the state of the interpreter, but also the code and the state of the interpreted program. Conditional breakpoints may be expressed with respect to both the interpreter and the interpreted program. In that respect, JSExplain is a double-debugger for the specification of JavaScript.

**ACM Reference Format:**
Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *WWW '18 Companion: The 2018 Web Conference Companion, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3184558.3185969

## 1 INTRODUCTION

### 1.1 A reference interpreter for JS

JavaScript (JS) has a complex semantics. As of 2017, its specification by ECMA consists of 885 pages of English prose [6] (details in §1.2). Unsurprisingly, this prose-based presentation of the specification does not meet the needs of the JavaScript designers and implementers. In particular, the JavaScript standardization committee (TC39) has repeatedly expressed the need for better tools for describing and interacting with the semantics (§1.3).

Prior work on the formalization of JS semantics, notably JSCert [3] and KJS [11], has made some progress, yet falls short of delivering several of the features needed by TC39 (§1.4). In this work, we aim at addressing these requests: we revisit the JSCert semantics by giving it a presentation more accessible to the JavaScript community. Our presentation aims to be well-suited for writing and reading the specifications, executing test cases, checking coverage, and interatively debugging the specification (§1.5).

The JS specification is essentially describing a reference interpreter. Although it consists of English prose, the ECMA standard reads almost like pseudo-code. Most ambiguities and unclear paragraphs that were present in ECMA3 and ECMA5 were progressively resolved in subsequent editions. Thus, turning ECMA pseudo-code into real code, i.e., code expressed in a real programming language, is not so hard. Yet, there are two nontrivial aspects: dealing with the representation of the state, and dealing with abrupt termination, such as exceptions, return, break, and continue statements.


*WWW'18 Companion, April 23–27, 2018, Lyons, France*

ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3185969

Indeed, in JS, the evaluation of any sub-expression, of any type conversion, and of most internal operations from the specification may result in the execution of user code, hence the raising of an exception, interrupting the normal control flow. Throught its successive editions, the ECMA standard progressively introduced a notation akin to an exception monad (§1.2). This notation is naturally translated into real code by a proper monadic bind operator of the exception monad.

Regarding the state, the standard assumes a global state. A reference interpreter could either assume a global state, modified with side-effects, or thread the state explicitly in purely-functional style. We chose the latter approach for three reasons. First, we already need a monad for exceptions, so we may easily extend this monad to also account for the state. Second, starting from code with an explicit state would make it easier to generate a corresponding inductive definition in a formal logic (e.g., Coq), which we would like to investigate in the future. Third, to ease the reading, one may easily hide a state that is explicitly threaded; the converse, materializing a state that is implicit, would be much more challenging.

We thus write our reference interpreter in a purely-functional language extended with syntactic sugar for the monadic notation to account for the state and the propagation of abrupt termination (§2). For historical reasons, we chose a subset of OCaml as source syntax, but other languages could be used. In fact, we implemented a translator from our subset of OCaml to a subset of JS (a subset involving no side effects and no type conversions). We thereby obtain a JS interpreter that is able to execute JS programs inside a JS virtual machine—JS fans should be delighted. To further improve accessibility to JS programmers, we also translate the source code of our interpreter into a human-readable JS-style syntax, which we call pseudo-JS, and that essentially consists of JS syntax augmented with a monadic notation and with basic pattern matching.

Our reference semantics for JS is inherently executable. We may thus execute our interpreter on test suites, either by compiling and executing the OCaml code, or by executing the JS translation of that code. It is indeed useful to be able to check that the evaluation of examples from the JS test suites against our reference semantics produces the desired output.

Even more interesting is the possibility to investigate, step by step, the evaluation of the interpreter on a given test case. Such investigation allows to understand *why* the evaluation of a particular test case produces a particular output—given the complexity of JS, even an expert may easily get puzzled by the output value of a particular piece of code. Furthermore, interactive execution makes it easier for the contributor of a new JS feature to add new test cases and to check that these tests trigger the new features and correctly interact with existing features.

In this paper, we present a tool, called JSExplain, for investigating JS executions. This tool can be thought as a *double debugger*, which

displays both the state of the interpreted program and that of the interpreter program. In particular, our tool supports conditional breakpoints expressed simultaneously on the interpreter program and the interpreted program. To implement this tool, we generate a version of our interpreter that is instrumented for producing execution traces (§3), and we provide a web-based tool to navigate through such traces (§4). As far as we know, our tool is the first such double debugger, i.e., debugger with specific additions for dealing with programs that interpret other programs (§5).

## 1.2 English Specification of JS

To illustrate the style in which the JavaScript standard (ECMA) [6] is written, consider the description of addition, which will be our running example throughout the paper. In JS, the addition operator casts its arguments to integers and computes their sum, except if one of the two arguments is a string, in which case the other argument is cast to a string and the two strings are concatenated.

The ECMA5 presentation (prior to June 2016) appears in Figure 1. First, observe that the presentation describes both the parsing rule for addition and its evaluation rule. Presumably for improved accessibility, the JS standard does not make explicit the notion of an abstract syntax tree (AST). The semantics of addition goes as follows: first evaluate the left branch to a value, then evaluate the right branch to a value, then converts both values (which might be objects) into primitive values (e.g., string, number, ...), then test whether one of the two arguments is a string. If so, cast both arguments to strings and return their concatenation; otherwise cast both arguments to numbers and return their sum.

This presentation style used in ECMA5 gives no details about the propagation of exceptions. While the treatment of exceptions is explicit for statements, it is left implicit for expressions. For example, if the evaluation of the left branch raises an exception, the right branch should not be evaluated. It appeared that leaving the treatment of exceptions implicit could lead to ambiguities at what exactly should or should not be evaluated when an exception gets triggered. The ECMA committee hates such ambiguities, because it could (and typically does) result in different browsers exhibiting different behaviors—the nightmare of web-developers.

In ECMA6, such ambiguities were resolved by making the propagation of exceptions explicit. Figure 2 shows the updated specification for the addition operator. There are two main changes. First, each piece of evaluation is described on its own line, thereby making the evaluation order crystal clear. Second, the meta-operation ReturnIfAbrupt is invoked on every intermediate result. This meta-operation essentially corresponds to an exception monad. The ECMA6 standards, which aims to be accessible to a large audience, avoids the introduction of the word "monad". Instead, it specifies ReturnIfAbrupt as a "macro", as shown in Figure 3. Essentially, every result consists of a "completion record", which corresponds to a sum type distinguishing normal results from exceptions.

In all constructs except try-catch blocks, exceptions interrupt the normal flow of the evaluation. As a result, ECMA6 specification is scattered with about 1100 occurences of ReturnIfAbrupt operations. Realizing the impracticability of that style of specification, the standardization committee decided to introduce in ECMA7 an additional layer of syntactic sugar in subsequent versions of the

**Evaluation of:** *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *lprim* be ToPrimitive(*lval*).
6. Let *rprim* be ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   - Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*).
8. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*).

**Figure 1: ECMA5 specification of addition.**

**Evaluation of:** *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).
9. Let *rprim* be ToPrimitive(*rval*).
10. ReturnIfAbrupt(*rprim*).
11. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. let *lstr* be ToString(*lprim*).
    b. ReturnIfAbrupt(*lstr*).
    c. let *rstr* be ToString(*rprim*).
    d. ReturnIfAbrupt(*rstr*).
    e. Return the String that is the result of concatenating *lstr* and *rstr*.
12. let *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. let *rnum* be ToNumber(*rprim*).
15. ReturnIfAbrupt(*rnum*).
16. Return the result of applying the addition operation to *lnum* and *rnum*.

**Figure 2: ECMA6 specification of addition.**

specification. As detailed in Figure 4, they define the question mark symbol to be a lightweight shorthand for ReturnIfAbrupt steps. The specification of addition in that new style is shown in Figure 5.

The presentation of ECMA7 and ECMA8 (Figure 5) is both more concise than that of ECMA6 (Figure 2) and more formal than that of ECMA5 (Figure 1). The use of question marks is to be compared in §2 with the monadic notation that we use for our formal semantics.

## 1.3 Requests from the JS Committee

The JavaScript standardization body, part of ECMA and known as TC39, includes representatives from browser vendors, major

**Evaluation of:** ReturnIfAbrupt

Algorithms steps that say

1. ReturnIfAbrupt(*argument*).

mean the same thing as:

1. If *argument* is an abrupt completion, return *argument*.
2. Else if *argument* is a Completion Record, let *argument* be *argument*.[[value]].

**Figure 3: ECMA6 interpretation of ReturnIfAbrupt.**

Algorithms steps that say or are otherwise equivalent to:

1. ReturnIfAbrupt(AbstractOperation()).

mean the same thing as:

1. Let *hygienicTemp* be AbstractOperation().
2. If *hygienicTemp* is an abrupt completion, return *hygienicTemp*.
3. Else if hygienicTemp is a Completion Record, let *hygienicTemp* be *hygienicTemp*.[[Value]].

Where *hygienicTemp* is ephemeral and visible only in the steps pertaining to ReturnIfAbrupt.

Invocations of abstract operations and syntax-directed operations that are prefixed by **?** indicate that ReturnIfAbrupt should be applied to the resulting Completion Record. For example, the step:

1. **?** OperationName().

is equivalent to the following step:

1. ReturnIfAbrupt(OperationName()).

**Figure 4: ECMA7 and ECMA8 addition for ReturnIfAbrupt.**

actors of the web, and academics. It aims at defining a common semantics that all browsers should implement. TC39 faces major challenges. On the one hand, it must ensure full backward compatibility, to avoid "breaking the web". In particular, no feature used in the wild ever gets removed from the specification. On the other hand, the committee imposes the rule that no feature may be added to the standard before it has been implemented, shipped, and tested at scale in at least two distinct major browsers. Any member of the committee may propose new features, hence there are many proposals being actively developed, at different stages of formalization [14].

The rapid evolution of the standard stresses the need for appropriate tools to assist in the rewriting, testing, and debugging of the semantics. In particular, several members with whom we have had interactions expressed their need for several basic tools, such as:

- a tool for knowing whether all variables that occur in the specification are properly defined (bound) somewhere;
- a tool to perform basic type-checking of the meta-functions and of the variables involved in the specification;
- a tool for checking that effectful operations go on a line of their own, to avoid ambiguity in the order of evaluation;
- a tool for checking that the behavior is specified in all cases;

**Evaluation of:** *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   a. let *lstr* be ? ToString(*lprim*).
   b. let *rstr* be ? ToString(*rprim*).
   c. Return the String that is the result of concatenating *lstr* and *rstr*.
8. let *lnum* be ? ToNumber(*lprim*).
9. let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*.

**Figure 5: ECMA7 and ECMA8 specification of addition.**

- a tool able to tell which lines from the specification are not covered by any test from the main test suite (test262 [15]);
- a tool able to execute step-by-step the specification on concrete JS programs, and to inspect the value of the variables.

In particular, step-by-step execution is critically needed to evaluate new features. When the committee decides that a feature proposal is worth integrating, it typically does not accept the proposal as is, but instead modifies the proposal in a way that is amenable to a simple, clear specification without corner cases, carefully trying to avoid harmful interactions with other existing features (or planned features). During this process, at some point the committee members have in their hand a draft of the extended semantics as well as a collection of test cases illustrating the new behaviors that should be introduced. Naturally, they would like to check that their formalization of the extended semantics assigns the expected behavior to each of the test cases.

One might argue that such a task could be performed by modifying one of the mainstream browsers. Yet, existing JS runtimes are built with efficiency in mind, with huge code bases involving numerous optimizations. As such, modifying the code in any way is too costly for committee members to investigate variations on a feature request. Even if they could invest the effort, the distance between the English prose specification and the implementation would be too large to have any confidence that the two match, i.e., that the behavior implemented in the code matches the behavior described by the prose.

An alternative approach to testing a new feature is to develop an elaboration (local translation) of that feature into plain JS. This can take the form of syntactic sugar adding a missing API, namely a *polyfill*, or the form of a source to source translation, namely a *transpiler*. For instance, one might translate so-called "template literals" into simple string concatenation.

```
/* new feature */
var name = "me";
`hello ${name}`;
```

```
/* plain JavaScript */
var name = "me";
"hello " + name;
```

While polyfills and transpilers are a simple approach to testing new features, they have two major limitations. First, the encoding might be very invasive. For instance, the 2015 version of ECMAScript added *proxies*, and as a consequence significantly changed the internal methods of the language; the Babel [1] transpiler for proxies [2] is able to simulate this feature in prior version of JS, but at the cost of replacing all field access operations with calls to wrapper functions. Second, the interaction of several new features implemented using these approaches is very difficult to anticipate.

### 1.4 Formal Specifications of JS

In recent years, two projects, JSCert [3] and KJS [11] have proposed a formal specification for a significant subset of JS. JSCert provides a big-step inductive definition for ECMA5, (technically, a pretty-big-step specification [5]), formalized in the Coq proof assistant [16]. JSCert comes along with a reference interpreter, called JSRef, that is proved correct with respect to the inductive definition. JSRef may be extracted into executable OCaml code for executing tests. KJS describes a small-step semantics for JS as a set of rewriting rules, using the K framework [13]. This framework has been used to formalize the semantics of several other real-world languages. It provides in particular tool support for executing (syntax-directed) transition rules on a concrete input program.

At first sight, it might seem that a formal specification addresses all the requests from the committee. Definitions are thoroughly type-checked; in particular, all variables must be properly bound. Definitions, being defined in a formal language, are ambiguity-free; in particular, the order of evaluation and the propagation of exceptions is precisely specified. The semantics can be executed on concrete input programs; moreover, with some extra tooling, one may execute a set of programs and report on the coverage of the specification by the tests.

Given all the nice features of formal semantics, why wouldn't the standardization committee TC39 consider one of these formal semantics as the reference for the language? After discussing with senior members from TC39, we understand that there are (at least) three main reasons why there is no chance for a formal semantics such as JSCert or KJS to be adopted as reference semantics.

(1) Formal specifications in Coq or in K use syntax and concepts that are not easily accessible to JS programmers. Yet, the specification is meant to be read by a wide audience.
(2) These formal languages have a cost of entry that is too high for committee members to reach the level of proficiency required for contributing new definitions all by themselves.
(3) JSCert and KJS come with specifications that can be executed, yet provide no debugger-style support for interactively navigating through an execution and for visualizing the state and the values of the variables, and thus do not help so much in tuning the description of new features.

In the present work, we temporarily leave aside the motivation of giving a formal semantics to JS that one could use to formalize properties of the language (e.g., security properties), and rather focus on trying to provide a formal semantics that meets better the day-to-day needs of the TC39 committee.

### 1.5 Contribution

In this paper, we present a tool, called JSExplain, which aims at providing a formal semantics for JS that addresses the aforementioned limitations of prior work. Our contribution is two-fold. First, we present a specification for JS expressed in a language that, we argue, JS programmers can easily read and write (§2). Second, we present an interactive tool that supports step-by-step execution of the specification on an input JS program (§3 and §4). Our tool mimics the features of a debugger, such as navigation controls, state and variable visualization, and conditional breakpoints, but does so for both the interpreter program and the interpreted program.

The language in which we display the specification consists of a subset of JS extended with syntactic sugar for monads and basic pattern matching. This language, which we call pseudo-JS, could be the source syntax for our specification. However, for historical and technical reasons, we use as input syntax a subset of OCaml, which is processed using the OCaml type-checker. Our current tool automatically converts the OCaml AST into pseudo-JS code. In the future, we might as well have our reference interpreter be directly in pseudo-JS syntax, and we could typecheck that code either by converting it to OCaml or by reimplementing a basic ML type-checker. A third alternative would be to use the Reason syntax [12], a JS-like syntax for OCaml programs. The only difference between the approaches is whether TC39 committee members would prefer to write OCaml style or JS style code.

## 2 SPECIFICATION LANGUAGE

### 2.1 Constructs of the Language

The input syntax in which we write and display the specification is a purely-functional language that includes the following constructs: variables, constants, sequence, conditional, let-binding, function definition, function application (with support for prefix and infix functions), data constructors, records (including record projections, and the "record-with" construct to build a copy of a record with a number of fields updated), tuples (i.e., anonymous records), and simple pattern matching (only with non-nested patterns, restricted to data constructors, constants, variables, and wildcards). For convenience, let-bindings and functions may bind patterns (as opposed to only variables).

We purposely aim for a specification language with a limited number of constructs and a very standard semantics, to minimize the cost of entry into that language. Note that the input code is type-checked in ML. (Polymorphism is used mainly for type-checking options and lists, and operations on them.)

As explained earlier (§1.2), the semantics involves the propagation of exceptions and other abrupt behaviors (break, continue, and return). Their propagation can be described within our small language, using functions and pattern matching. Nevertheless, introducing a little bit of syntactic sugar greatly improves readability. For example, we write "`let%run x = e1 in e2`" to mean "`if_run e1 (fun x -> e2)`", where `if_run` is a function that implements our monad.

The monadic operator `if_run` admits a polymorphic type, hence functions from the specification may return objects of various types. Nevertheless, in practice most functions from the ECMA standard

```
and run_binary_op s c op v1 v2 =
  match op with
  | C_binary_op_add -> run_binary_op_add s c v1 v2
  ...
and run_binary_op_add s0 c v1 v2 =
  let%prim (s1, w1) = to_primitive_def s0 c v1 in
  let%prim (s2, w2) = to_primitive_def s1 c v2 in
  if  (type_compare (type_of (Coq_value_prim w1)) Coq_type_string)
   || (type_compare (type_of (Coq_value_prim w2)) Coq_type_string)
  then
    let%string (s3, str1) = to_string s2 c (Coq_value_prim w1) in
    let%string (s4, str2) = to_string s3 c (Coq_value_prim w2) in
    res_out (Coq_out_ter (s4, (res_val (Coq_value_prim (Coq_prim_string (strappend str1 str2))))))
  else
    let%number (s3, n1) = to_number s2 c (Coq_value_prim w1) in
    let%number (s4, n2) = to_number s3 c  (Coq_value_prim w2) in
    res_out (Coq_out_ter (s4, (res_val (Coq_value_prim (Coq_prim_number (n1 +. n2))))))
```

**Figure 6: Current input syntax of our specification language: a subset of pure OCaml, extended with monadic notation.**

are described as returning a "completion triple", which either describe abrupt termination or describe a value. In a number of cases, the value is in fact constrained to be of a particular type. For example, if `to_number` produces a value, then this value is necessarily a number. The standard exploits this invariant implicitly in formulation such as "let *n* be the number produced by calling `to_number`". In constrast, our code needs to explicitly project the number from the value returned. To that end, we introduce specialized monads such as `if_number`, written in practice "`let%number n = e1 in e2`". (An alternative approach would be to assign polymorphic types to completion triples, however following this route would require diverging slightly from ECMA's specification in a number of places.)

Figure 6 shows the specification of addition in our reference interpreter, in OCaml syntax extended with the monadic notation. This code implements its informal equivalent from Figure 2. In that code, `s` denotes the state, `c` denotes the environment (variable and lexical environment, in JS terminology), `op` corresponds to the operator (here, the constructor `C_binary_op_add` corresponds to the AST token describing the operator `+`), `v1` and `v2` corresponds to the arguments, and `w1` and `w2` to their primitive values. The function `strappend` denote string concatenation, whereas "`+.`" denotes addition on floating pointer numbers (i.e., JS's numbers).

First observe that, as explained earlier (§1.1), the state is threaded throughout the code. We show in the next section how to hide the state variables (§2.2). Observe also that the code also relies on a few auxiliary functions. The function `type_compare` implements comparison over JS types—to keep our language small and explicit, we do not want to assume a generic comparison function with nontrivial specification. The functions `to_primitive_def`, `to_string`, and `to_number` are internal functions from the specification that implement conversions. These operations might end up evaluating arbitrary user code, and thus could perform side-effects or raise exceptions, hence the need to wrap them in monadic let-bindings.

One important feature of this source language is that it does not involve any "implicit" mechanism. All type conversions are explicit in the code, so it is always perfectly clear what is meant. In particular, there is no need to type-check the code to figure out its semantics. In summary, the OCaml code of the interpreter (e.g., Figure 6) is well-suited as a non-ambiguous input language. Note that this code may be compiled using OCaml's compiler in order to run test cases; the current version of our interpreter passes more than 5000 test cases from the official test suite (test262).

```
var run_binary_op = function (op, v1, v2) {
 switch (op) {
  case C_binary_op_add:
    return (run_binary_op_add(v1, v2));
  ... }
};
var run_binary_op_add = function (v1, v2) {
  var%prim w1 = to_primitive_def(v1);
  var%prim w2 = to_primitive_def(v2);
  if ((type_compare(type_of(w1), Type_string)
      || type_compare(type_of(w2), Type_string))) {
    var%string str1 = to_string(w1);
    var%string str2 = to_string(w2);
    return strappend(str1, str2);
  } else {
    var%number n1 = to_number(w1);
    var%number n2 = to_number(w2);
    return (n1 + n2);
  }
};
```

**Figure 7: Generated code for the interpreter in pseudo-JS syntax, with implicit environments, state, and casts.**

## 2.2 Translation into Pseudo-JS Syntax

Although we believe that it is a desirable feature to have a source langage fully explicit, there is also virtue in pretty-printing the source code of our interpreter in a more concise syntax. The "noise" that appears in the formal specification (e.g. Figure 6) comes from three main sources:

(1) every function takes as argument the environment;
(2) every function takes as argument and returns a description of the mutable state (a.k.a. heap);

(3) values are typically built using numerous constructors, e.g. `C_value_prim`, which lifts a number (an OCaml value of type `float`) into a JS value (an OCaml value of type `value`).

Fortunately, we can easily eliminate these three sources of noises.

First, the environment is almost always passed unchanged. It may be modified only during the scope of a function call, a with construct, or a block. When it is modified, new bindings are simply pushed into the environment (which behaves like a stack), and subsequently popped. Thus, we may assume, like the ECMA specification does, that the environment is stored in a global state. This saves the need to pass an argument called "`c`" around.

Second, the description of the mutable state is threaded through the code. The "current state" is passed as argument to every function that might perform side-effects, and, symmetrically, the "updated state" is returned to the caller, which binds a fresh name for it. Considering that there is only one version of the state at any given point of an execution, we may assume, like the ECMA specification does, that state to be stored in a global variable. This saves the need to pass values called "`s1`", "`s2`"... around.

Third, the presence of many constructors is due to the need for casts. Many of these casts could, however, be viewed as "implicit casts" (or "coercions", in Coq's terminology). For a carefully chosen set of casts, defined once and for all, and for a well-typed program with implicit casts, there exists a unique (non-ambiguous) way to insert casts in order to make the program type-check. Although we have previously argued that explicit casts are useful, as they allow giving a semantics that does not depend on type-checking, we now argue that it may also be useful to pretty-print the code assuming implicit casts, in order to improve readability.

In summary, we propose to the reader of the specification a version that features implicit state, implicit context, and implicit casts. Given that we are playing the game of pretty-printing syntax, we take the opportunity to switch along the way to a JS-friendly syntax, using brackets and semicolons. This target language, called pseudo-JS syntax, consists of a subset of the JS syntax, extended with monadic notation, and an extended `switch` construct that is able to bind variables (like OCaml's pattern matching, but restricted to non-nested patterns for simplicity).

The pretty-printing of the addition operator in pseudo-JS syntax appears in Figure 7. To illustrate our extended pattern matching syntax feature of pseudo-JS, we show below an excerpt from the main switch that interprets an expression.

```
switch (t) {
 case Coq_expr_identifier(x):
   var%run r = identifier_resolution(x); return (r);
 case Coq_expr_binary_op(e1, op, e2): ...
```

## 3 TRACE-PRODUCING EXECUTIONS

JSExplain is a tool for interactively investigating execution traces of our JS interpreter executing example JS programs. The interface consists of a web page [8] that embeds a JS parser and a trace-producing version of our interpreter implemented in standard JS.

So far, we have shown how to translate the OCaml source into pseudo-JS syntax (§2.2). In this section, we explain how to translate the OCaml source into proper JS syntax, and then how to instrument the JS code in a systematic way for producing execution traces.

Figure 8 illustrates the output of translating from our OCaml subset towards JS. Note that this code is not meant for human consumption. We implement monadic operators as function calls, introduce the `return` keyword where necessary, encode sum types as object literals with a `tag` field, encode tuples as arrays (encoding tuples as object literals would work too), turn constructor applications into functions calls, implement pattern matching by first switching on the `tag` field then binding fresh variables to denote the arguments of constructors.

We thus obtain an executable JS interpreter in JS which, like our JS interpreter in OCaml, may be used for executing test cases. One interest of the JS version is that it may be easily executed inside a browser, a set up that might be more convenient for a number of users. One limitation, however, is that the number of steps that can be simulated may be limited on JS virtual machines that do not optimize tail-recursive function calls. Indeed, the execution of monadic code involves repeated calls to continuations, whose (tail-call) invocation unnecessarily grows the call stack.

To set up our interactive debugger, we produce, from our OCaml source code, an instrumented version of the JS translation. This instrumented code produces execution traces as a result of interpreting an input JS program. These traces store information about all the states that the interpreter goes through. In particular, each event in the trace provides information about the code pointer and the instantiations of local variables from the interpreter code.

More precisely, we log events at every entry point of a function, every exit point, and on every variable binding. Each event captures the state, the stack, and the values of all local variables in scope of the interpreter code at the point where the event gets triggered. To reduce noise in the trace, we only log events in the core code of the interpreter, and not the code from the auxiliary libraries. Overall, an execution of the instrumented interpreter on some input JS program produces an array of events. This array can then be investigated using our double debugger (§4).

Figure 9 shows an example snippet of code, giving an idea of the mechanisms at play. Note, again, that this code is not meant for human consumption. The function `log_event` augments the trace. Consider for instance `log_event("Main.js", 4033, ctx_747, "enter")`. The first two arguments identify the position in the source file, as a file name and a unique token used to recover the line numbers. The third argument is a context describing values of the local variables, and the fourth argument describes the type of event.

When investigating the trace, we need to be able to highlight the corresponding line of the interpreter code. We wish to be able to do so for the three versions of the interpreter code: the OCaml version, the pseudo-JS version, and the plain JS version. To implement this feature, our generator, when processing the OCaml source code, also produces a table that maps, for each version and for each file of the interpreter, event tokens to line numbers.

The contexts stored in events are extended each time a function is entered, a new variable is declared, or the function returns (so as to capture the returned value). Contexts are represented as a purely-functional linked list of mappings between variable names and values. This representation maximizes sharing and thus minimize the memory footprint of the generated trace. The length of the trace grows linearly with the number of execution steps performed. For example, the simple program "`var i = 0; while (i < N) { i++ }`"

```
var run_binary_op_add = function (s0, c, v1, v2) {
  return (if_prim(to_primitive_def(s0, c, v1), function(s1, w1) {
    return (if_prim(to_primitive_def(s1, c, v2), function(s2, w2) {
      if ((type_compare(type_of(Coq_value_prim(w1)), Coq_type_string())
        || type_compare(type_of(Coq_value_prim(w2)), Coq_type_string())) {
          return (if_string(to_string(s2, c, Coq_value_prim(w1)), function(s3, str1) {
            return (if_string(to_string(s3, c, Coq_value_prim(w2)), function(s4, str2) {
              return (res_out(Coq_out_ter(s4, res_val(
                    Coq_value_prim(Coq_prim_string(strappend(str1, str2)))))); }));}));
      } else { ... }})); }));
};
```

**Figure 8: Snippet of generated code for the interpreter in standard JS syntax, without trace instrumentation.**

```
var run_binary_op_add = function (s0, c, v1, v2) {
  var ctx_747 = ctx_push(ctx_empty, [{key: "s0", val: s0}, {key: "c", val: c}, {key: "v1", val: v1}, {key: "v2", val: v2}]);
  log_event("JsInterpreter.js", 4033, ctx_747, "enter");
  var _return_1719 = if_prim((function () {
      log_event("JsInterpreter.js", 3985, ctx_747, "call");
      var _return_1700 = to_primitive_def(s0, c, v1);
      log_event("JsInterpreter.js", 3984, ctx_push(ctx_747, [{key: "#RETURN_VALUE#", val: _return_1700}]), "return");
      return (_return_1700); }()),
    function(s1, w1) { ... });
   log_event("JsInterpreter.js", 4028, ctx_push(ctx_748, [{key: "#RETURN_VALUE#", val: _return_1718}]), "return");
   return (_return_1718);
   });
  log_event("JsInterpreter.js", 4032, ctx_push(ctx_747, [{key: "#RETURN_VALUE#", val: _return_1719}]), "return");
  return (_return_1719);
};
```

**Figure 9: Snippet of generated code for the interpreter in standard JS syntax, with trace instrumentation.**

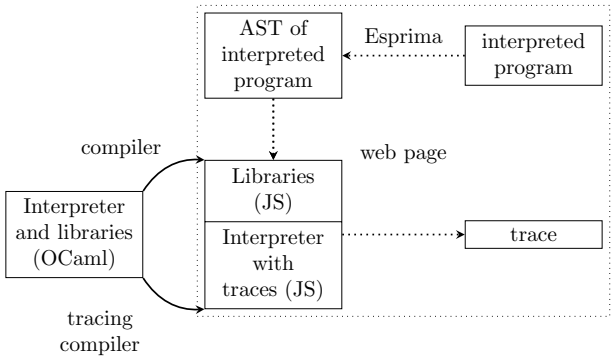


**Figure 10: Architecture of JSExplain.**

generates a trace of size 2 990 for $N = 1$, of size 14 166 for $N = 10$, of size 126 126 for $N = 100$, and of size 1 245 726 for $N = 1000$.

The fact that these numbers are large reflects the fact that the reference interpreter is inherently vastly inefficient, as it follows the specification faithfully, without any optimization. Due to our use of functional data structures, the memory footprint of the trace should be linear in the length of the trace. We have not observed the memory footprint to be a limit, but if it were we could more carefully select which events should be stored.

## 4 JSEXPLAIN: A DOUBLE DEBUGGER FOR JS

The global architecture of JSExplain is depicted in Figure 10. Starting from our JS interpreter in OCaml, we generate a JS interpreter in JS. We instrument the JS code to produce a trace of events. This compilation is done ahead of time and depicted by solid arrows.

When hitting the `run` button, the flow depicted by the dotted arrows occurs. The web page parses the code from the text area, using the Esprima library [7]. This parser produces an AST, with nodes annotated with locations. This AST is then provided as input to the instrumented interpreter, which generates a trace of events. This trace may then be inspected and navigated interactively.

For a given event from the execution trace, our interface highlights the corresponding piece of code from the interpreter, and shows the values of the local variables, as illustrated in Figure 11. It also highlights the corresponding piece of code in the interpreted program, as illustrated at the top of Figure 13, and displays the state and the environment of the program at that point of the execution, as illustrated in Figure 12.

Recovering the information about the interpreted code is not completely straightforward. For example, to recover the fragment of code to highlight, we find in the trace the closest previous event that contains a call to function with an argument named `_term_`. This argument corresponds to the AST of a subexpression, and this AST is decorated (by the parser) with locations. Note that, for efficiency reasons, we associate to each event from the trace its corresponding `_term_` argument during a single pass, performed immediately after the trace is produced.

Similarly, we are able to recover the state and environment associated with the event. The state of the interpreted program consists of four fields: the strictness flag, the value of the `this` keyword,

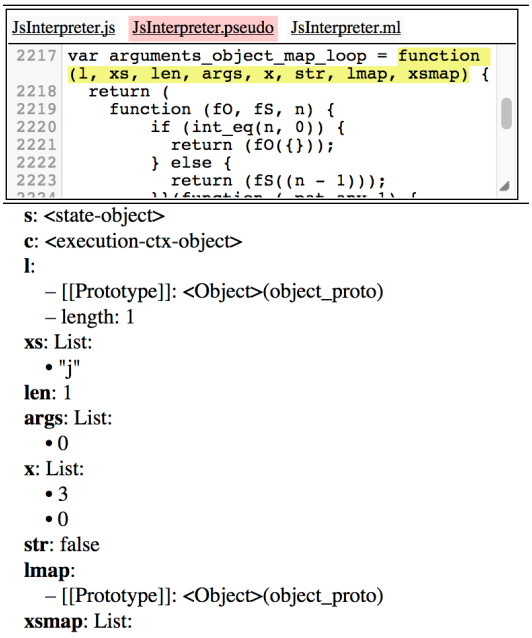


**Figure 11: Display of the variables from the interpreter code**

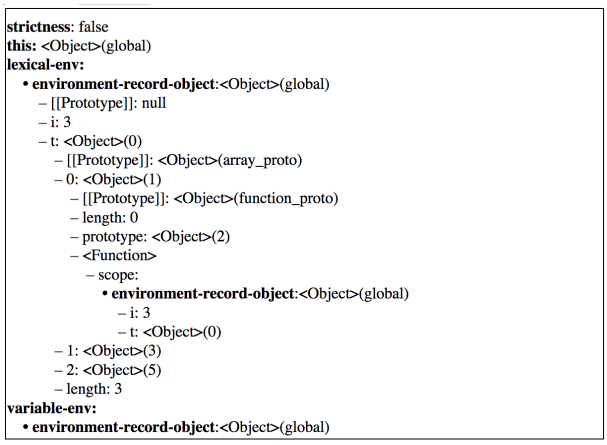


**Figure 12: Display of the state and environment of the interpreted code. The environment includes the local variables.**

the lexical environment, and the variable environment. We implemented a custom display for these elements, and also for values of the languages, in particular for objects: one may click on an object to reveal its contents and recursively explore it.

We provide several ways to navigate the trace. First, we provide buttons for reaching the beginning or end of the execution, and buttons for stepping one event at a time. Second, we provide, similarly to debuggers, *next* and *previous* buttons for skipping function calls, as well as a *finish* button to reach the end of the current function. These features are implemented by navigating the trace, keeping track of the number of *enter* and *return* events. Third, we provide buttons for navigation based not on steps related to the interpreter program but instead based on steps of the interpreted program: *source previous* and *source next* find the closest event which induces

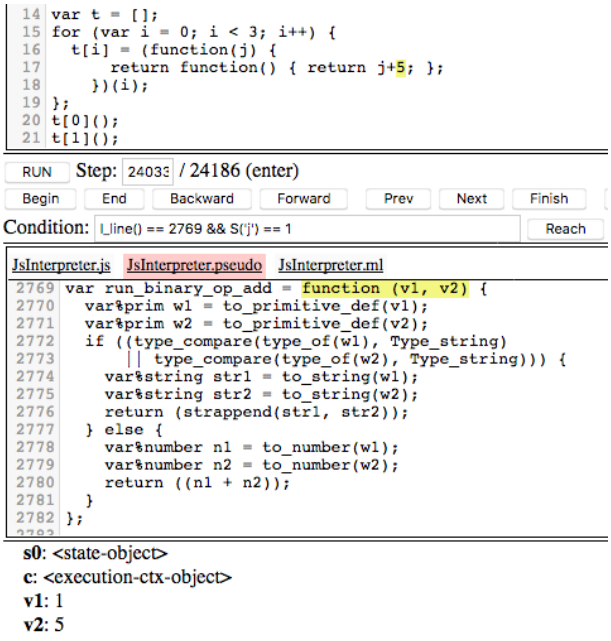


**Figure 13: Example of a conditional breakpoint, constraining the state of both the interpreter and the interpreted code.**

a change in the location on the subexpression evaluated in the interpreted code, and *source cursor* finds the last event in the trace for which the associated subexpression contains the active cursor in the "source program" text area.

The aforementioned tools are sufficient for simple explorations of the trace, yet we have found that it is sometimes useful to reach events at which specific conditions occur, such as being at a specific line in the interpreter, in the interpreted code, with variables from the interpreter or interpreted code having specific values. We thus provide a text box to enter arbitrary breakpoint conditions to be evaluated on events from the trace. For example, the condition in Figure 13 reaches the next occurrence of a call to `run_binary_op_add` in a context where the source variable `j` has value 1. The break point condition may be any JS expression using the following API: `I_line()` returns the current line of the interpreter, `S_line()` returns the current line of the source, `I('x')` returns the value of `x` in the interpreter, `S_raw('x')` returns the value of `x` in the source (e.g. the JS object `{tag: "value_number", arg: 5}`), and `S('x')` returns the JS interpretation of the value of `x` in the source (e.g. the JS value `5`).

## 5 RELATED WORK

There are many formal semantics of JavaScript, from pen-and-paper ones [10], to the aforementioned JSCert [3] and KJS [11]. As described in §1.4, these semantics are admirable but lack crucial features to be actively used in the standardization effort.

To our knowledge, the closest work to the double-debugger approach is the multi-level debugging approach of Kruck et al. [9]. They present a debugger for an interpreter for domain-specific languages that lets developers choose the level of abstraction at which they debug their program. An abstraction is a way to display some values (encoded in the host language, or as present in the DSL) as well as showing only stack frames that represent computation

at the DSL level. Our technique is more general as it does not focus on domain-specific languages.

In fact, our double-debugger approach could be easily adapted to interpreter for other languages than JS. To that end, it suffices to implemented an interpreter for the desired language in the subset of OCaml that we support, and to provide code for extracting and displaying the term and state associated with a given event. We have recently followed that approach and adapted our framework to derive a double-debugger for (a significant subset of) the OCaml programming language.

Regarding the translation from OCaml to JS that we implement, one might consider using an existing, general-purpose tool. Js_of_ocaml [17] converts OCaml bytecode into efficient JS code. Presumably, we could implement the logging instrumentation as an OCaml source-to-source translation and then invoke Js_of_ocaml. Yet, with that approach, we would need to convert the representation of trace events from the encoding of these values performed by Js_of_ocaml into proper JS objects that we can display in the interactive interface. This conversion is nontrivial, as some information, such as the name of constructors, is lost in the process. As we already implemented a translator from OCaml to pseudo-JS, it was simpler to implement a translator from OCaml to plain JS.

Another translator from OCaml to JS is Bucklescript [4], which was released after we started our work. Similarly to our translator, Bucklescript converts OCaml code into JS code advertised as readable. Bucklescript also has the limitation that the names of constructors are lost, although presumably this could be easily fixed. Besides, if we wanted to revisit our implementation to base it on Bucklescript, for trace generation we would need to either modify Bucklescript, which is quite complex as it covers the full OCaml language, or to reimplement trace instrumentation at the OCaml level, which should be doable yet would involve a bit more work than at the level of untyped JS code.

## 6 CONCLUSION AND FUTURE WORK

We presented JSExplain to TC39[1] and the committee expressed strong interest. They would like us to extend our specification to cover all of the specification. We have almost finished the formalization of proxies, which are a challenging addition to the language as they change many internal methods. Although all members seem to agree that the current toolset for developing the specification is inappropriate, it requires a strong leadership and a consensus to commit to a new toolchain. Our goal is to cover the current version of ECMAScript, we currently cover ECMAScript 5, and to help committee members use it to formalize new additions to JavaScript.

There are numerous directions for future work. (1) We plan to set up a modular mechanism for describing unspecified behaviors (e.g. "for-in" enumeration order) as well as browser-specific behaviors (sometimes browsers deviate from the specification, for historical reasons). (2) We could investigate the possibility of extending the formalization of the standard by also covering the parsing rules of JS; currently, our semantics is expressed with respect to the AST of the input program. (3) To re-establish a link with the original JSCert inductive definition, which is useful for conducting formal proofs about the metatheory of the langage, we would like to investigate the possibility of automatically generating pretty-big-step [5] definitions from the reference semantics expressed in our small language, possibly using some amount of annotation to guide the process. (4) To close even further the gap between a formal language and the English prose, we could also investigate the possibility of automatically generating English sentences from the code. Indeed, the prose from the ECMAScript standard is written in such a systematic manner that this should be doable, at least to some extent.

[1] https://tc39.github.io/tc39-notes/2016-05_may-25.html#jsexplain-as--tw


## ACKNOWLEDGMENTS

We acknowledge funding from the ANR project AJACS ANR-14-CE28-0008 and the CominLabs project SecCloud.


## REFERENCES


[1] 2017. Babel. (October 25, 2017). https://babeljs.io/.

[2] 2017. Babel proxy plugin. (October 25, 2017). https://www.npmjs.com/package/babel-plugin-proxy.

[3] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised javascript specification. In *Proceedings of POPL 2014*, 87–100.

[4] 2017. Bucklescript. (October 27, 2017). https://bucklescript.github.io/bucklescript/.

[5] Arthur Charguéraud. 2013. Pretty-big-step semantics. In *Proceedings of ESOP 2013* (LNCS). Volume 7792. Springer, 41–60.

[6] ECMA. 2017. Ecmascript 2017 language specification (ecma-262, 8th edition). (June 2017). https://www.ecma-international.org/ecma-262/8.0/index.html.

[7] 2017. ECMAScript parsing infrastructure for multipurpose analysis. (October 26, 2017). http://esprima.org/.

[8] 2017. JSExplain. (October 26, 2017). https://jscert.github.io/jsexplain/branch/master/driver.html.

[9] Bastian Kruck, Stefan Lehmann, Christoph Keßler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. 2016. Multi-level debugging for interpreter developers. In *MODULARITY (Companion)*. ACM, 91–93.

[10] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An operational semantics for javascript. In *Proceedings of APLAS 2008* (LNCS). Volume 5356. Springer, 307–325.

[11] Daejun Park, Andrei Stefanescu, and Grigore Rosu. 2015. KJS: a complete formal semantics of javascript. In *Proceedings of PLDI 2015*. ACM, 346–356.

[12] 2017. Reason. (October 27, 2017). https://reasonml.github.io/.

[13] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79, 6, 397–434.

[14] 2017. TC39 proposals. (October 26, 2017). https://github.com/tc39/proposals.

[15] 2017. Test262. (October 26, 2017). https://github.com/tc39/test262.

[16] The Coq development team. 2014. *The Coq proof assistant reference manual*. Version 8.4. http://coq.inria.fr.

[17] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Exprerience*, 44, 8, 951–972.